An inherent bottleneck in distributed counting

An Inherent Bottleneck in Distributed Counting

Roger Wattenhofer Peter Widmayer

Institut für Theoretische Informatik
ETH Zürich, Switzerland

Abstract

A distributed counter allows each processor in an asynchronous message passing network to access the counter value and increment it. We study the problem of implementing a distributed counter such that no processor is a communication bottleneck. We prove a lower bound of $\Omega(k)$ on the number of messages that some processor must exchange in a sequence of n counting operations spread over n processors, where $kk^k = n$. We propose a counter that achieves this bound for the situation it is derived for, namely, when each processor increments the counter exactly once. Hence, the lower bound is tight. Because most algorithms and data structures count in some way, the lower bound holds for many distributed computations. We feel that the proposed concept of a communication bottleneck is a relevant measure of efficiency for a distributed algorithm and data structure, because it indicates the achievable degree of distribution.

1 Introduction

Counting is an essential ingredient in virtually any computation. It is therefore highly desirable to implement counters efficiently. In a distributed setting, efficiency has a number of important constituents, such as time or message complexity. Various precise measures of efficiency for these constituents have been established in the literature; for instance, the time complexity of a distributed algorithm in an asynchronous setting measures the worst case time from the start of a run to its completion, based on the assumption that each message takes only one time unit.

One important aspect of efficiency, however, is mostly taken into account only on an intuitive basis in the construction of distributed algorithms and data structures: The work of the algorithm should not be concentrated at any single processor or within a small group of processors, even if this optimizes some measure of efficiency. For instance, even though a data structure implementing a distributed counter could be message optimal by just storing the counter value with a single processor and having all other processors access the counter with only one message exchange, such an implementation is clearly unreasonable: This solution does not scale – whenever a large number of processors operate on the counter, the single processor handling the counter value will be a bottleneck.

This paper studies the bottleneck that is inherent in any counting mechanism in a distributed, asynchronous setting. We characterize this bottleneck by deriving a lower bound on the number of messages that some processor must handle in a sequence of operations. Even in the simple case where test-and-increment is the only supported operation, the lower bound holds. It carries over directly to any data structure in which the effect of an operation critically depends on the preceding operation(s) – that is, virtually any other data structure. We also propose a distributed

counter with an optimum bottleneck for the specific counting problem that we use in our lower bound proof. This shows that the lower bound is tight.

Related Work

To the best of our knowledge, this is the first study of the inherent bottleneck in a distributed data structure. Therefore, in the literature there are no close relatives to this paper. Two tracks of research in distributed computation, however, relate loosely to our work, namely distributed counters and quorum systems.

Efficient implementations of a distributed counter received considerable attention in the past few years. The Combining Trees proposed in [YTL86] and in [GVW89] were the first to explicitly aim at avoiding a bottleneck. Based on bitonic sorting networks, [AHS91, HLS92] developed Counting Networks; they have been analyzed in [DHW93, HSW96]. Diffracting Trees by [SZ94] enjoy the benefits of Combining Trees and Counting Networks; they are analyzed and tested in [SUZ96].

Some of the reasoning in our paper is closely related with that in quorum systems. A quorum system is a collection of sets of elements, where every two sets in the collection intersect. The foundations of quorum systems have been laid in [GB85] and [Mae85]; [Nei92] contains a good survey on the topic. A dozen ways have been proposed of how to construct quorum systems, starting in the early seventies [Lov73, EL75] and continuing until today [HMP95, PW95, KM96, PW96]. Even though the approach we present in this paper might be called a "Dynamic Quorum System", it has nothing in common with [JM90].

2 Distributed Counting: The Model

Consider an asynchronous, distributed system of n processors in a message passing network, where each processor is uniquely identified with one of the integers from 1 to n. Each processor has unbounded local memory; there is no shared memory. Any processor can exchange messages directly with any other processor. There is no a priori bound on the length of a message. A message arrives at its destination an unbounded, but finite amount of time after it has been sent. No failures whatsoever occur in the system.

An abstract data type *distributed counter* is to be implemented for such a distributed system. A distributed counter encapsulates an integer value *val* and supports the operation *inc* for any processor. *inc* returns the current counter value *val* to the requesting processor, and increments the counter (by one). For the sake of deriving a lower bound, we will ignore concurrency control problems, that is, we will not propose a mechanism that synchronizes messages and local operations at processors. Let us therefore assume that enough time elapses in between any two *inc* requests to make sure that the preceding *inc* operation is finished before the next one starts.

An *inc* operation (request) initiates a process, i.e., a partially ordered set of events in the distributed system. Let us examine the process of a single *inc* operation. Let p be the processor that initiates the *inc* operation. To do so, p sends a message to several other processors; these in turn send messages to others, and so on. After a finite amount of time, p receives the last of the messages which lets p determine the current value *val* of the counter (for instance, p may simply receive the counter value in a message). This does not necessarily terminate the *inc* process: Additional messages may be sent in order to prepare for future operations. As soon as

no further messages are sent, the *inc* process terminates. During this process, the counter value has been incremented.

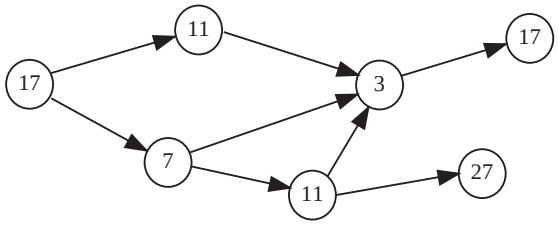

Figure 1: Processor 17 initiates an *inc* operation

We can visualize the process of an *inc* operation as a directed, acyclic graph (DAG). A node with label q of the DAG represents processor q performing some communication. In general, a processor may appear more than once as a node label; in particular, the initiating processor p appears as the source of the DAG and somewhere else in the DAG, where p is informed of the current counter value *val*. An arc from a node labelled p_1 to a node labelled p_2 denotes the sending of a message from processor p_1 to processor p_2. For initiating processor p, let I_p denote the set of all processors that send or receive a message during the observed *inc* process. The following lemma appears in similar form in many papers on quorum systems [Mae85].

Hot Spot Lemma: Let p and q be two processors that increment the counter in direct succession. Then $I_p \cap I_q \neq \emptyset$ must hold.

Proof: For the sake of contradiction, let us assume that $I_p \cap I_q = \emptyset$. Because only the processors in I_p can know the current value *val* of the counter after p's *inc* process, none of the processors in I_q knows about the *inc* operation initiated by p. Therefore, q gets an incorrect counter value, a contradiction. □

Note that the argument in the Hot Spot Lemma can be made for the family of all distributed data structures in which an operation depends on the operation that immediately precedes it. Examples for such data structures are a bit that can be accessed and flipped, and a priority queue. In this paper, however, we restrict our attention to the distributed counter.

The process of an *inc* operation is not a function of the requesting processor and the state of the system, due to the nondeterministic nature of a distributed computation. Here, the state in between any two operations is defined as a vector of the local states of the processors. Now consider a prefix of the DAG of a process *proc* in state s of the system, and consider the set *pref* of processors present in that prefix. Then for any state of the system that is different from s, but is identical when restricted to the processors in *pref*, the considered prefix of *proc* is a prefix of a possible process. The reason is simply that the processors in *pref* cannot distinguish between both states, and therefore any partially ordered set of events that involves only processors in *pref* and can happen nondeterministically in one state can happen in the other as well.

3 A Lower Bound on the Message Load

Definitions: Consider a sequence of consecutive *inc* operations of a distributed counter. Let m_p denote the number of messages that processor p sends or receives during the operation sequence;

we call this the *message load* of processor p. Choose a processor b with $m_b = max_{p=1..n} m_p$ and call b a *bottleneck processor*.

We will derive a lower bound on the load for the interesting case in which not too many operations are initiated by any single processor. One can easily show that the amount of achievable distribution is limited if many operations are initiated by a single processor. To be even more strict for the lower bound, we request that each processor initiates exactly one *inc* operation.

Lower Bound Theorem: In any algorithm that implements a distributed counter on n processors, there is a bottleneck processor that sends and receives $\Omega(k)$ messages, where $kk^k = n$.

Proof: To simplify the argument, let us replace the communication DAG of an *inc* process by a topologically sorted, linear list of the nodes of the DAG. This communication list models the DAG so that each message along an arc in the DAG corresponds to a sequence of messages along a path in the list. By counting each arc in the list just once, we get a lower bound on the number of messages per processor in the DAG, because no processor has more incoming arcs to nodes with its label in the list than in the DAG.

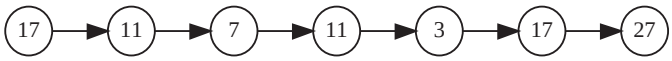

Figure 2: Example of Figure 1 as a list

Now let us study a particular sequence of n *inc* operations, where each processor initiates one operation. For each operation in the sequence, there may be more than one possible process. We will argue on possible prefixes of processes for each of the operations. The sequence of operations is defined as follows: For each operation in the sequence we choose a processor (among those that have not been chosen yet) and a process such that the processor's communication list is longest, where the length is measured as the number of arcs in the list. Let processor i denote the processor that is chosen for the i-th operation in the sequence and let L_i be the length of the chosen communication list of processor i. Thus, the number of messages that are sent for the i-th operation is exactly L_i in the list. For the total of n *inc* operations, the number of messages sent is $\sum_{i=1}^{n} L_i$; let us denote this number as $n \cdot L$, with L as the average number of messages sent. Because every sent message is going to be received by a processor, we have $\sum_{p=1}^{n} m_p = 2nL$. This guarantees the existence of a processor b with $m_b \geq \lceil \frac{2nL}{n} \rceil \geq 2L$.

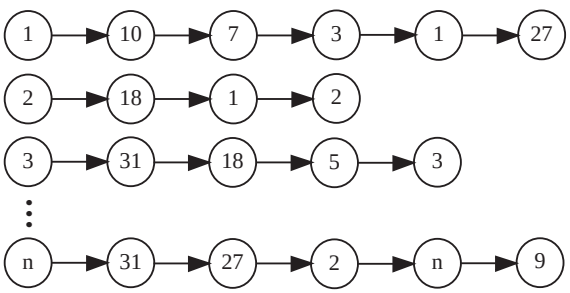

Figure 3: Situation before initiating an *inc* operation

We will now argue on the choice of a processor for the ith *inc* operation. This choice is made according to the lengths of the communication lists of processors that have not incremented yet,

see Figure 3. We will compare the list lengths of the chosen processor and the processor that is chosen only for the very last *inc* operation. Let q be this last processor in the sequence, and consider the list of processor q for the i-th *inc* operation. Let l_i denote the length of this list. By definition of the operation sequence, $l_i \leq L_i$ $(i = 1, \ldots, n)$. Let $p_{i,j}$ denote the processor label of the j-th node of the list, where $j = 0, 1, \ldots, l_i$. By definition of the communication list, we have $p_{i,0} = q$ for $i = 1, \ldots, n$.

Now define the weight w_i for the i-th *inc* operation as

$$w_i := \sum_{j=0}^{l_i} \frac{m(p_{i,j})}{\mu^j},$$

where $m(p_{i,j})$ is the number of messages that processor $p_{i,j}$ sent or received before the i-th *inc* operation, and $\mu := m_b + 1$. Initially, we have $m(p) = 0$ for each $p = 1, \ldots, n$, and therefore $w_1 = 0$.

How do the weights of the two consecutive lists of processor q differ when an *inc* operation is performed? To see this, let us compare the weights w_i and w_{i+1}. The Hot Spot Lemma tells us that at least one of the processors in q's list must receive a message in the i-th *inc* operation; let $p_{i,f}$ be the first node with that property in the list. The list for *inc* operation $i+1$ can differ from the list for *inc* operation i in all elements that follow $p_{i,f}$, including $p_{i,f}$ itself, but there is at least one process in which it is identical to the list before *inc* operation i in all elements that precede $p_{i,f}$ (formally, $p_{i+1,j} = p_{i,j}$ for $j = 0, \ldots, f$). The reason is that for none of the processors preceding $p_{i,f}$, the (knowledge about the system) state changes due to the i-th *inc* operation.

This immediately gives

$$\begin{aligned}
w_{i+1} &= w_i + \frac{1}{\mu^f} + \sum_{j=f+1}^{l_{i+1}} \frac{m(p_{i+1,j})}{\mu^j} - \sum_{j=f+1}^{l_i} \frac{m(p_{i,j})}{\mu^j} \\
&\geq w_i + \frac{1}{\mu^f} - \sum_{j=f+1}^{l_i} \frac{m(p_{i,j})}{\mu^j} \\
&\geq w_i + \frac{1}{\mu^f} - \sum_{j=f+1}^{l_i} \frac{\mu - 1}{\mu^j} \\
&= w_i + \frac{1}{\mu^f} - (\frac{1}{\mu^f} - \frac{1}{\mu_i^l}) \\
&= w_i + \frac{1}{\mu^{l_i}}
\end{aligned}$$

We therefore have

$$w_n \geq \sum_{i=1}^{n-1} \frac{1}{\mu^{l_i}}$$

Processor q sent and received at least $m(p_{n,0})$ messages in the sequence of n *inc* operations. We have

$$m(p_{n,0}) = w_n - \sum_{j=1}^{l_n} \frac{m(p_{n,j})}{\mu^j}$$

With $\mu - 1 = m_b \geq m_q \geq m(p_{n,0})$ we get

$$\begin{aligned}
\mu &\geq w_n - \sum_{j=1}^{l_n} \frac{m(p_{n,j})}{\mu^j} + 1 \\
&\geq w_n - \sum_{j=1}^{l_n} \frac{\mu - 1}{\mu^j} + 1 \\
&= w_n - (1 - \frac{1}{\mu^{l_n}}) + 1 \\
&= w_n + \frac{1}{\mu^{l_n}} \\
&\geq \sum_{i=1}^{n} \frac{1}{\mu^{l_i}} \\
&\geq n \sqrt[n]{\prod_{i=1}^{n} \frac{1}{\mu^{l_i}}} \\
&= n \sqrt[n]{\mu^{-\sum_{i=1}^{n} l_i}} \\
&\geq n \sqrt[n]{\mu^{-\sum_{i=1}^{n} L_i}} \\
&= n \sqrt[n]{\mu^{-nL}} \\
&= \frac{n}{\mu^L} \qquad (1)
\end{aligned}$$

That is, $\mu \geq \sqrt[L+1]{n}$. With $\mu > m_b \geq 2L > L$ we conclude $\mu > k$, where $kk^k = n$. Since $m_b = \mu - 1$, this proves the claimed lower bound. □

4 A Matching Upper Bound

We propose a distributed counter that achieves the lower bound of the previous section in the worst case. It is based on a communication tree whose root holds the counter value. The leaves of the tree are the processors that request *inc* operations. The inner nodes of the tree serve the purpose of forwarding an *inc* operation request to the root. Recall that each of the n processors requests exactly one *inc* operation.

The communication tree structure is as follows. Each inner node in the communication tree has k children. All leaves of the tree are on level $k+1$; the root is on level zero. Hence the number of leaves is kk^k. For simplicity, let us assume that $n = kk^k$ (otherwise, simply increase n to the next higher value of the form kk^k, for integer k).

Each inner node in the tree stores $k+3$ values. It has an identifier *id* that tells which processor currently works for the node; let us call this the current processor of the node. For simplicity, we do not distinguish between a node and its current processor whenever no ambiguity arises. Furthermore, it knows the identifiers of its k children and its parent. In addition, it keeps track of the number of messages that the node sent or received since its current processor works for it; we call this its *age*.

Initially, node j $(j = 0, \ldots, k^i - 1)$ on level i $(i = 1, \ldots, k)$ gets the identifier

$$(i-1)k^k + jk^{k-i} + 1.$$

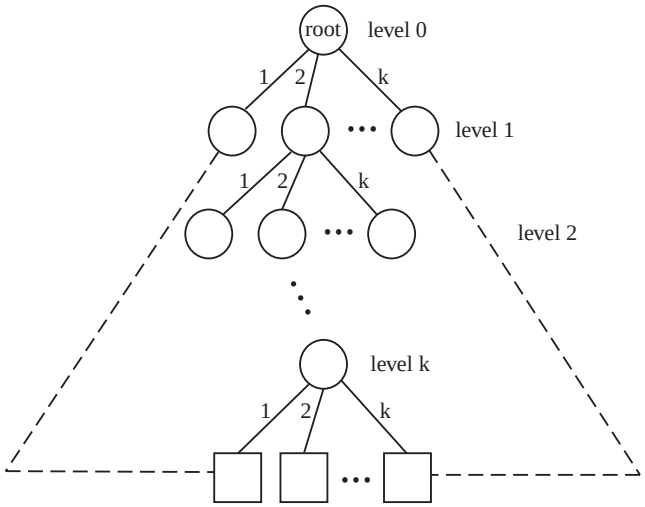

Figure 4: Communication Tree Structure

Numbered this way, no two inner nodes on levels 1 through k get the same identifier. Furthermore, the largest identifier (used for the parent of the rightmost leaf) has the value

$$(k-1)k^k + (k^k - 1)k^{k-k} + 1 = kk^k - k^k + k^k - k^0 + 1 = kk^k = n.$$

We will make sure that no two inner nodes on levels 1 through k ever have the same identifiers. The root, nevertheless, starts with $id = 1$. The leaves have identifiers $1, 2, \ldots, n$ from left to right on level $k+1$, representing the n processors. Since all ids are defined by this regular scheme, all the processors can compute all initial identifiers locally. The age of all inner nodes including the root is initially 0. The root stores an additional value, the counter value val, where initially $val = 0$.

The inc Operation

Now let us describe how an inc operation initiated at processor p is carried out. The leaf whose id is p sends a message <inc from p> to its parent. Any non-root node receiving an <inc from p> message forwards this message to its parent and increments its age by two (one for receiving and one for sending a message). When the root receives an <inc from p> message, it sends a message $<val>$ to processor p and then increments val; furthermore, it increments its age by two. After incrementing its age value, a node decides locally whether it should retire: It will retire if and only if it has $age \geq 4k$. To retire, the node updates its local values by setting $age := 0$ und $id_{new} := id_{old} + 1$; it then sends $2k+2$ final messages. $k+1$ messages inform the new processor of its new job and the of the ids of its parent and children nodes, the other $k+1$ messages inform the node's parent and children about id_{new}. Note that in this way, we're able to keep the length of messages as short as $O(\log n)$ bits. There is a slight difference when the root retires: It additionally informs the new processor of the counter value val, and it saves the message that would inform the parent. Since the parent and children nodes receive a message, they increment their age values. It may of course happen that this increment triggers the retirement of parent and children nodes. If so, they again inform their parent, their children and the new processor as described. For simplicity, we do not describe here the details of handling the corresponding messages; one way of solving this problem is a proper handshaking protocol, with a constant number of extra messages for each of the messages we describe.

The message load

While correctness is straightforward and is therefore omitted, we will now derive a bound on the message load in detail.

Retirement Lemma: No node retires more than once during any single *inc* operation.

Proof: Assume to the contrary that there is such a node, and let u be the first node (in historic order) that retires a second time. Since u is first, all children and the parent of u retired only once during the current *inc* operation. Therefore, u receives at most $k+1$ messages. Since $k+1 < 4k$ for $k \geq 1$ node u cannot retire twice. □

Grow Old Lemma: If an inner node does not retire during an *inc* operation, it sends and receives at most four messages.

Proof: Let p be the processor that initiates the *inc* operation. Each inner node u that is on the path from leaf p to the root receives one message from its child v on that path, and it forwards this message to its parent. Among all nodes adjacent to u, only its parent and v can retire during the current *inc* operation, because u's other children are not on the path from p to the root and belong to I_p only if u retires. Due to the Retirement Lemma no node can retire more than once during a single *inc* operation, thus u does not receive more than two retirement messages. To sum up, a node u receives one message if its parent retires and not more than three further messages if u is on the path from p to the root. □

Number of Retirements Lemma: During the entire sequence of n *inc* operations, each node on level i retires at most $k^{k-i} - 1$ times.

Proof: The root lies on each path and therefore receives at most two messages per *inc* operation and sends one message (the counter value). It retires after every $4k$ messages, with the total number r_0 of retirements satisfying

$$r_0 \leq \frac{3n}{4k} = \frac{3}{4}k^k < k^k.$$

In general, a node on level i is on k^{k-i+1} paths, and it receives and sends at most $3k^{k-i+1} + r_{i-1}$ messages. With a retirement at every $4k$ messages, we inductively get a total for the number r_i of retirements of a node on level i:

$$r_i \leq \frac{1}{4k}(3k^{k-i+1} + r_{i-1}) < \frac{1}{4k}(3+1)k^{k-i+1} = k^{k-i}.\Box$$

Let us now consider the availability of processors that replace others when nodes retire. The initial *id*'s at inner nodes on levels 1 through k have been defined just for the purpose of providing a sufficiently large interval of replacement processor identifiers. The jth node ($j = 0, \ldots, k^i - 1$) on level i ($i = 1, \ldots, k$) initially uses processor $(i-1)k^k + jk^{k-i} + 1$; its replacement processor candidates are those with identifiers

$$(i-1)k^k + jk^{k-i} + \{2, 3, \ldots, k^{k-i}\}.$$

Note that these are exactly $k^{k-i} - 1$ processors, just as needed in the worst case. In addition, note that the root replaces its processor $k^k - 1$ times.

Inner Node Work Lemma: Each processor receives and sends at most $O(k)$ messages while it works for a single inner node.

Proof: When a processor starts working for a node, it receives $k+1$ messages from its predecessor that tells about the identifiers of its parent and its children. From the Grow Old Lemma we conclude that it receives and sends at most $4k$ messages before it retires. Upon its retirement, it sends $k+1$ messages to its successor, and one to its parent and to each of its children. □

Leaf Node Work Lemma: During the entire sequence of n *inc* operations, each leaf receives and sends at most 2 messages.

Proof: Each leaf initiates exactly one *inc* operation and receives an answer, accounting for two messages. It receives an extra message whenever its parent retires. Since the parent is on level k, the Retirement Lemma tells us that this happens

$$k^{k-k} - 1 = k^0 - 1 = 0$$

times. □

Bottleneck Theorem: During the entire sequence of n *inc* operations, each processor receives and sends at most $O(k)$ messages, where $kk^k = n$.

Proof: Each processor starts working at most once for the root and at most once for another inner node. From the Number of Retirements Lemma and the Inner Node Work Lemma we conclude that the load for this part is at most $O(k)$ messages. From the Leaf Node Work Lemma we get two additional messages, with a total of $O(k)$ messages as claimed. □

Acknowlegements

We'd like to thank Masafumi "Mark" Yamashita and Thomas Roos for helpful discussions and pointers to the literature.

References

[AHS91] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks and multiprocessor coordination. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 348–358, New Orleans, Louisiana, 6–8 May 1991.

[DHW93] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 174–183, San Diego, California, 16–18 May 1993.

[EL75] Paul Erdös and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In *Infinite and Finite Sets*, pages 609–627, 1975.

[GB85] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Boston, Massachusetts, 3–6 April 1989. ACM Press.

[HLS92] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Low contention load balancing on large-scale multiprocessors. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 219–227, San Diego, California, June 29–July 1, 1992. SIGACT/SIGARCH.

[HMP95] Ron Holzman, Yosi Marcus, and David Peleg. Load balancing in quorum systems. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures (WADS'95)*, volume 955 of *LNCS*, pages 38–49, Berlin, GER, August 1995. Springer.

[HSW96] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *DISTCOMP: Distributed Computing*, 9, 1996.

[JM90] Sushil Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems; ACM CR 9102-0068*, 15(2), June 1990.

[KM96] Akhil Kumar and Kavindra Malik. Optimizing the costs of hierarchical quorum consensus. *Acta Informatica*, 33(3):255–275, 1996.

[Lov73] László Lovász. Coverings and colorings of hypergraphs. In *Proc. 4th Southwestern Conf. Combinatorics, Graph Theory and Computing*, pages 3–12, 1973.

[Mae85] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[Nei92] Mitchell L. Neilsen. *Quorum Structures in Distributed Systems.* PhD thesis, Dept. Computing and Information Sciences, Kansas State University, 1992.

[PW95] David Peleg and Avishai Wool. Crumbling walls: A class of practical and efficient quorum systems. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 120–129. ACM, August 1995.

[PW96] David Peleg and Avishai Wool. How to be an efficient snoop, or the probe complexity of quorum systems (extended abstract). In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 290–299. ACM, May 1996.

[SUZ96] Nir Shavit, Eli Upfal, and Asaph Zemach. A steady state analysis of diffracting trees. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 33–41, Padua, Italy, June 24–26, 1996. SIGACT/SIGARCH.

[SZ94] Nir Shavit and Asaph Zemach. Diffracting trees. In *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, pages 167–176, New York, NY, USA, June 1994. ACM Press.

[YTL86] Pen-Chung Yew, Niau-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large scale multiprocessor. In *International Conference on Parallel Processing*, pages 51–58, Los Alamitos, Ca., USA, August 1986. IEEE Computer Society Press.